USER MANUAL

<t-Base

Driver Developer's Guide

PREFACE

VERSION HISTORY

Version	Date	Status	Modification
1.0	26 Feb 2013	Issued	First Issued version
1.1	June 20th 2013	Corrected	Minor corrections

TABLE OF CONTENTS

TRUSTONIC

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

It is assumed here, that the reader is familiar with <t-base Application Developers Guide that applies to Trustlet development. In<t-base, Secure Drivers are similar to Trustlets and share many of the limitations, security considerations, and installation methods, as well as the build environment.

This guide is a practical introduction to:

- difference between Driver and Trustlet
- development of a <t-base Device Driver
- development of a corresponding Trustlet-Driver API

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

- Required: "MUST", "REQUIRED", "SHALL"
- Prohibited: "MUST NOT", "SHALL NOT"
- Implement whenever possible: "RECOMMENDED", “SHOULD”
- Avoid whenever possible: “NOT RECOMMENDED”, "SHOULD NOT"
- Nice to have/will be accepted: "OPTIONAL", "MAY"

2 ABBREVIATIONS

Table 1: Abbreviations

Abbreviation	Full name
DCI	Driver Control Interface
DDK	Driver Development Kit
DRAPI	Driver API
HAL	Hardware Abstraction Layer
IPC	Inter Process Communication
ISR	Interrupt Service Routine

TCI	Trustlet Control Interface
TLAPI	Trustlet API

3 What is a Device Driver?

A Device Driver enables a Trustlet to access HW resources in a controlled and coordinated manner. Driver must hide HW dependencies from Trustlet.

<t-base uses microkernel architecture. In microkernel architecture Secure Drivers are implemented as separate processes. Basically a Driver is a user mode task similar to a Trustlet or a System Trustlet. However it has different interface to <t-base that allows hardware access.

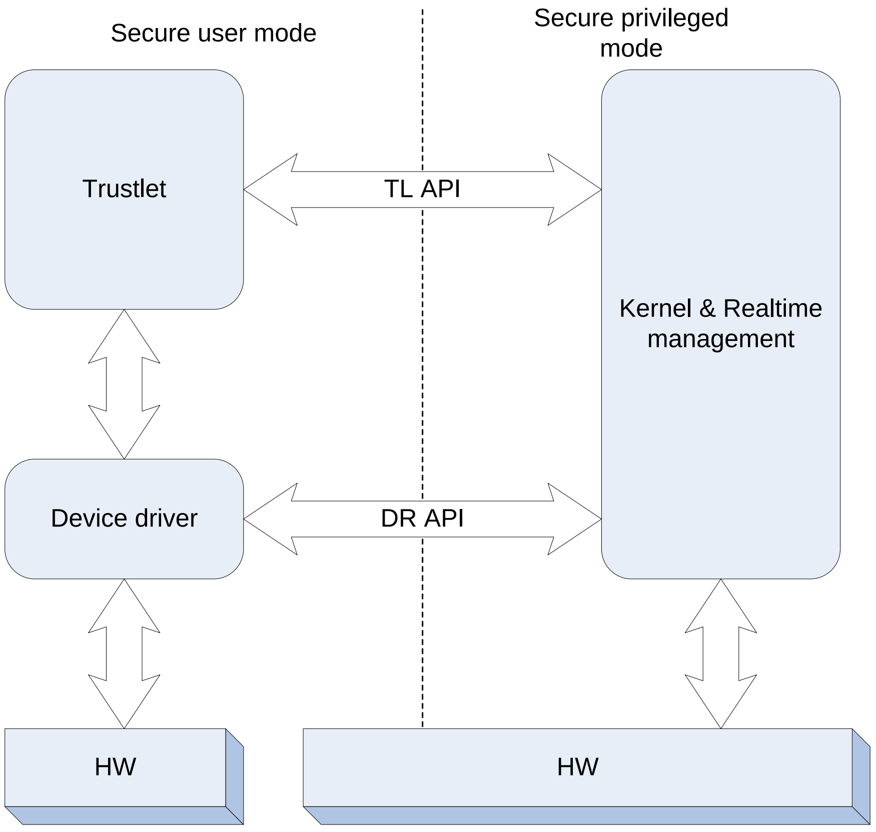

Figure 1: Simplified <t-base architecture

3.1 LOADING A DRIVER

<t-base supports dynamic loading of custom Drivers. Drivers are loaded and authenticated into <t-base in the same way as System Trustlets. Alternatively Drivers can be loaded by running <t-base normal world daemon with ‘-r’ option.

3.2 DRIVER INTERFACES

Drivers can use more system calls than Trustlets and thereby control hardware. The t-base system calls are available via the Driver API. Drivers cannot currently talk to each other.

Communication mechanism between Device Driver and Trustlet is message-passing IPC with marshalling of shared-memory.

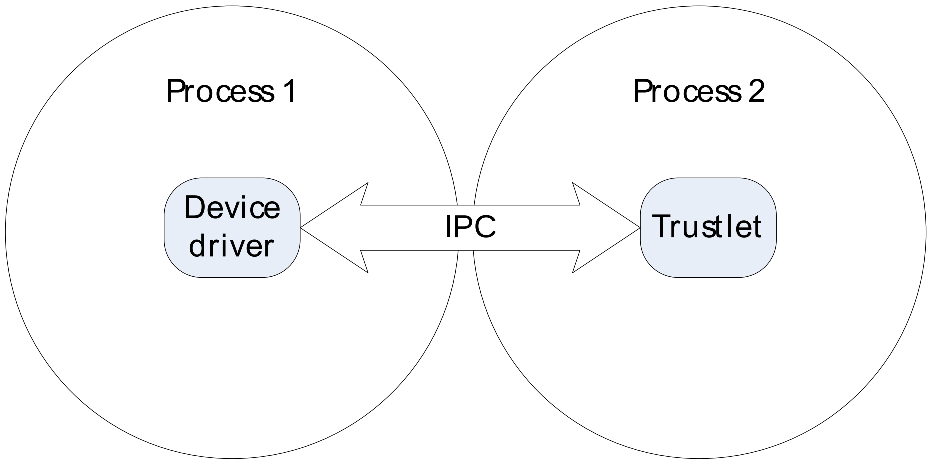

Figure 2: Device Driver and Trustlet

3.3 LIMITATIONS

There are number of limitations concerning the usage of Driver API:

- Trustlets cannot use Driver API, and cannot access hardware directly by doing so.
- Device Drivers cannot use Trustlet API and access <t-base crypto Driver.
- Drivers cannot talk to each other.

Because of these limitations, most use cases will have an architecture that consists of a TLC, a Trustlet and a Driver. The TLC will orchestrate everything and exchange encrypted objects with Trustlet. The Trustlet will decrypt them and pass commands to the Driver. The Driver will take commands from the Trustlet and transfer data in and out of the device.

3.4 DRIVER LIFE CYCLE

Drivers are dependent on <t-base version. No binary compatibility should be expected, as Drivers have significantly more access to <t-base internal details compared to Trustlets.

Drivers are security sensitive components, as they may be able to compromise <t-base security. Thus normally Trustonic will review source of all Drivers for proper implementation, including checking their security against malicious Trustlets.

3.5 DRIVER FEATURES AND SECURITY

3.5.1 Restricted Driver Processing Environment

- Device Driver runs in a special virtual address space that separates it from other Drivers, from Trustlets, from <t-base kernel and runtime, and from device memory.
- Drivers can communicate only via <t-base kernel system calls that are abstracted via Driver Api.

3.5.2 Memory Management

- To access device registers and peripheral memory, Driver can map physical memory into its virtual address space using Driver API.
- Driver can map physical memory to access registers and peripherals
- Driver can map memory regions and can do it with cached or uncached
- Driver can translate own virtual address to physical address by using a dedicated API mentioned below.
- Driver can do L1 d-cache clean/clean invalidate operations.
- To access data residing in Trustlet memory, Driver may map entire Trustlet memory to its virtual address space. This is required for parameter passing to and from a Trustlet. There is a function in Driver API that can be used for this purpose.

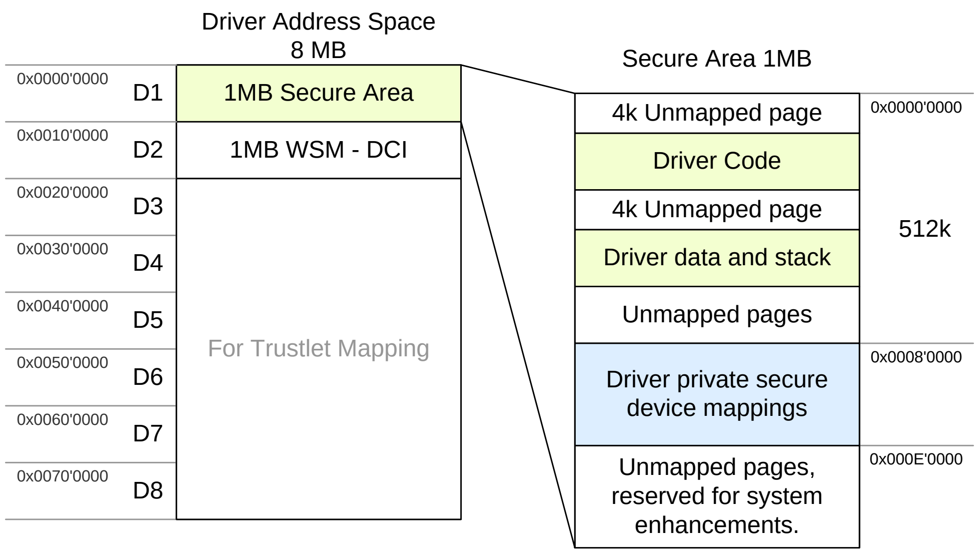

Figure 3: Driver Address Space

3.5.3 IPC

Communication between Trustlet and Driver must be done with IPC. Driver IPC API can only be used to pass notifications with one register of payload. Usually, this payload is a pointer to a buffer in Trustlet address space. Command IDs and data structures can be used to marshal call parameters onto this buffer.

3.5.4 Parameter checks in Drivers

- Device Driver must validate all data that it gets from Trustlet. For example pointers to memory areas must be verified.
- All given length parameters must be checked to not to exceed memory areas.

3.5.5 Interrupts, Threads, Exceptions and Messages

- Driver starts off with one thread.
- To wait for an IPC, Driver must call a blocking Driver API function.
- To wait for interrupts, Driver must call a blocking Driver API function.
- Driver can create multiple threads to allow waiting for various events.
- Driver can send IPC between threads for synchronization.
- In addition to IPC from Trustlets, Driver also has to handle system messages (e.g. Trustlet close messages received from <t-base)
- Driver can handle exceptions in its threads. Therefore an exception handler thread must be installed that waits for exceptions and restarts failing threads.

3.5.6 Reliability and Security

- Device Drivers are secure critical components and should not allow an attacker to compromise <t-base security
- Device Drivers should avoid complexity and large size.
- Device Drivers should react to all system messages appropriately and in a timely manner.
- Device Drivers should avoid polling.
- Device Drivers should not map in memory from <t-base.
- Device Drivers should not allow a Trustlet to access arbitrary memory.

3.5.7 Instances and Sessions

- There can be only one instance of each Driver.
- Driver exports a Driver ID, Trustlet can call this Driver ID.
- Driver can handle only one Trustlet at a time.
- Driver can implement sessions for Trustlets and use Trustlet ID.

4 Implementing Driver

Driver consists of:

- main thread that works as an exception handler thread
- IPC handler thread for handling IPC messages from Trustlets and system messages from <-base
- ISR handler thread if there is a need to attach to an interrupt and wait for that interrupt to occur

4.1 DRIVER MAIN THREAD/EXCEPTION HANDLER THREAD

Depending on the purpose of the Driver, the main thread should implement at least following functionalities:

- Initialization of the task/thread control
- Starting IPC and ISR handler threads
- Implementing exception handler loop for handling exceptions caused by other local threads and restarting them when possible
- Taking care of needed HW initialization
- Support for power saving mode whenever the task is idle

4.2 IPC HANDLER THREAD

The task of the IPC handler thread is to handle incoming IPC messages and process them accordingly.

When IPC thread starts, it needs to send ready message (‘MSG_RD’ notification) to <t-base (only when called first time) and then wait IPC messages by calling the following Driver API :

```
drApiResult_t drApiIpcCallToIPCH(
    threadid_t       *pIpcPeer,
    message_t        *pIpcMsg,
    uint32_t         *pIpcData
);
```

The following is an example of usage:

```
    for (;;)
    {
        /*
         * When called first time sends ready message to IPC server
and
         * then waits for IPC requests
         */
        if (E_OK != drApiIpcCallToIPCH(&ipcClient, &ipcMsg, &ipcData))
        {
            continue;
        }

        /* Dispatch request */
        switch (ipcMsg)
        {
            case MSG_CLOSE_TRUSTLET:
                /**
                 * Trustlet close message
                 */
                ipcMsg = MSG_CLOSE_TRUSTLET_ACK;
                ipcData = TLAPI_OK;
                break;
            case MSG_CLOSE_DRIVER:
                /**
                 * Driver close message
                 */
                ipcMsg = MSG_CLOSE_DRIVER_ACK;
                ipcData = TLAPI_OK;
                break;
            case MSG_GET_DRIVER_VERSION:
                /**
```

```
                  * Driver version message
                  */
                 ipcMsg = (message_t) TLAPI_OK;
                 ipcData = DRIVER_VERSION  ;
                 break;
             case MSG_RQ:

                 /* init tlRet value */
                 tlRet = TLAPI_OK;

                 /**
                  * Handle incoming IPC requests via TL API.
                 */
                 pMarshal=
(drMarshalingParam_ptr)drApiMapClientAndParams(
                                             ipcClient,
                                             ipcData);

                 if (pMarshal)
                 {
                     /* Process the request */
                     switch (pMarshal->functionId)
                     {
                     case FID_DR_SAMLE01:
                         /**
                          * Handle Sample01 request accordingly
                          */
                     default:
                         /* Unknown message has been received*/
                         tlRet = E_TLAPI_UNKNOWN_FUNCTION;
                         break;
                     }
                 }

                 /* Update response data */
                 ipcMsg  = MSG_RS;
                 ipcData = tlRet;
                 break;
             default:
                 /* Unknown message has been received*/
                 ipcMsg  = MSG_RS;
                 ipcData = E_TLAPI_DRV_UNKNOWN;
                 break;
         }
     }
```

The *DrApiIpcCallToIPCH* provides the Trustlet Task ID in pIpcPeer parameter and one word of payload from the Trustlet in the pIpcData parameter. Common usage of this parameter is to point to a data structure with encoded call parameters. Driver can then map Trustlet into its address space and access the data structure.

4.2.1 Accessing Trustlet data from marshaling parameters

Driver can map Trustlet memory into its own memory space with Driver API memory mapping functions. This allows Driver to interpret and use data referenced using pointers in marshaled structure.

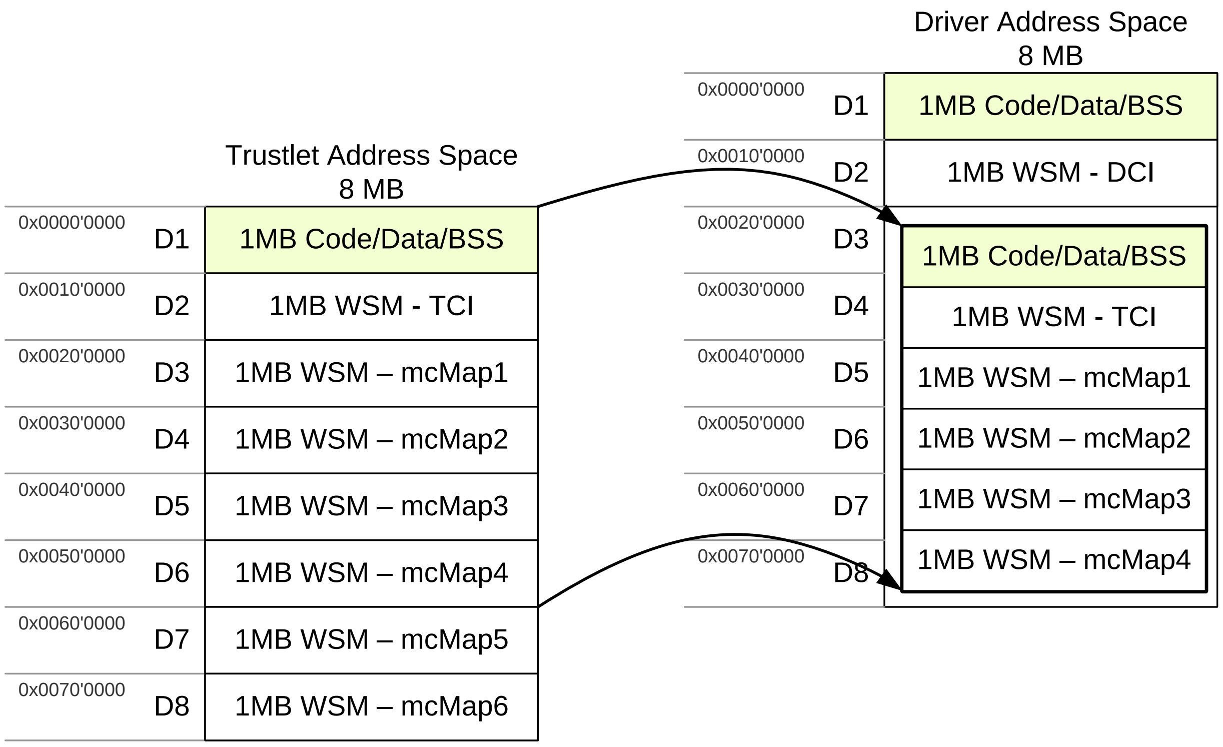
Trustlet Address Space
8 MB
0x0000'0000 D1 1MB Code/Data/BSS
0x0010'0000 D2 1MB WSM - TCI
0x0020'0000 D3 1MB WSM – mcMap1
0x0030'0000 D4 1MB WSM – mcMap2
0x0040'0000 D5 1MB WSM – mcMap3
0x0050'0000 D6 1MB WSM – mcMap4
0x0060'0000 D7 1MB WSM – mcMap5
0x0070'0000 D8 1MB WSM – mcMap6
Driver Address Space
8 MB
0x0000'0000 D1 1MB Code/Data/BSS
0x0010'0000 D2 1MB WSM - DCI
0x0020'0000 D3
0x0030'0000 D4
0x0040'0000 D5
0x0050'0000 D6
0x0060'0000 D7
0x0070'0000 D8
1MB Code/Data/BSS
1MB WSM - TCI
1MB WSM – mcMap1
1MB WSM – mcMap2
1MB WSM – mcMap3
1MB WSM – mcMap4

Figure 4: Mapping Trustlet into Driver

Parameters can be mapped from Trustlet memory space to Driver memory space with Driver API function:

```
drApiMarshalingParam_ptr drApiMapClientAndParams(
    threadid_t  ipcReqClient,
    uint32_t    params
);
```

Sample usage:

```
pMarshal= (drMarshalingParam_ptr)drApiMapClientAndParams(
                                                ipcClient,
                                                ipcData);
```

Trustlet data will have different address in Driver and in Trustlet. This means that all Trustlet data pointers must be converted before using them in the Driver. The following Driver API function can be used to convert Trustlet provided pointer to Driver address space. Function returns NULL, if pointer is not in correct range. Such pointer does not point to Trustlet virtual memory space.

```
 addr_t drApiAddrTranslateAndCheck(addr_t addr);
```

4.2.2 Unmapping Trustlet

When the Driver responds to Trustlet using *DrApiIpcCallToIPCH*, the IPCH will automatically unmap the Trustlet from the Driver.

4.3 HOW TO DO VIRTUAL/PHYSICAL ADDRESS TRANSLATION

Driver can translate a virtual address (from its own address space) to physical address by using the following Driver API:

```
drApiResult_t drApiVirt2Phys(
        const taskid_t      taskid,
        const addr_t        virtAddr,
        addr_t *            physAddr
);
```

As a result physical address is returned upon success.

4.4 HOW TO MAP PHYSICAL MEMORY

In order to access device registers, peripheral memory or any other memory region, Drivers map physical memory into its virtual address space. The following Driver API function can be used for that purpose:

```
drApiResult_t drApiMapPhys(
    const addr_t      startVirt,
    const uint32_t    len,
    const addr_t      startPhys,
    const uint32_t    attr
);
```

Drivers can do unmapping by using the following Driver API:

```
drApiResult_t drApiUnmap(
    const addr_t    startVirt,
    const uint32_t  len
);
```

The following is a sample usage of '*drApiMap()*' to map :

```
#define DR_VA_BUFFER (0x40000)
#define PHYS_MEM (0x12345678)

if (E_OK != drApiMapPhys(
            (page4KB_ptr) DR_VA_BUFFER,
            SIZE_4KB,
            (page4KB_ptr)(PHYS_MEM & ~(SIZE_4KB - 1)),
            MAP_READABLE | MAP_WRITABLE | MAP_UNCACHED))
    {
        /* Mapping failed */
    }
```

As a result 4KB memory starting from physical address *0x12345000* is mapped to virtual address *0x40000*.

Memory mapping attributes are listed below:

```
/* mapping does/shall have the ability to do read access. */
#define MAP_READABLE            (1U << 0)

/* mapping does/shall have the ability to do write access. */
```

```
#define MAP_WRITABLE                (1U << 1)

/* mapping does/shall have the ability to do program execution. */
#define MAP_EXECUTABLE              (1U << 2)

/* mapping does/shall have uncached memory access. */
#define MAP_UNCACHED                (1U << 3)

/* mapping does/shall have memory mapped I/O access.
#define MAP_IO                      (1U << 4)
```

Drivers should map devices into the designated device mappings area at address 0x00040000 - 0x00080000. Note that Driver has to manage its own virtual address space in respect to mappings several devices.

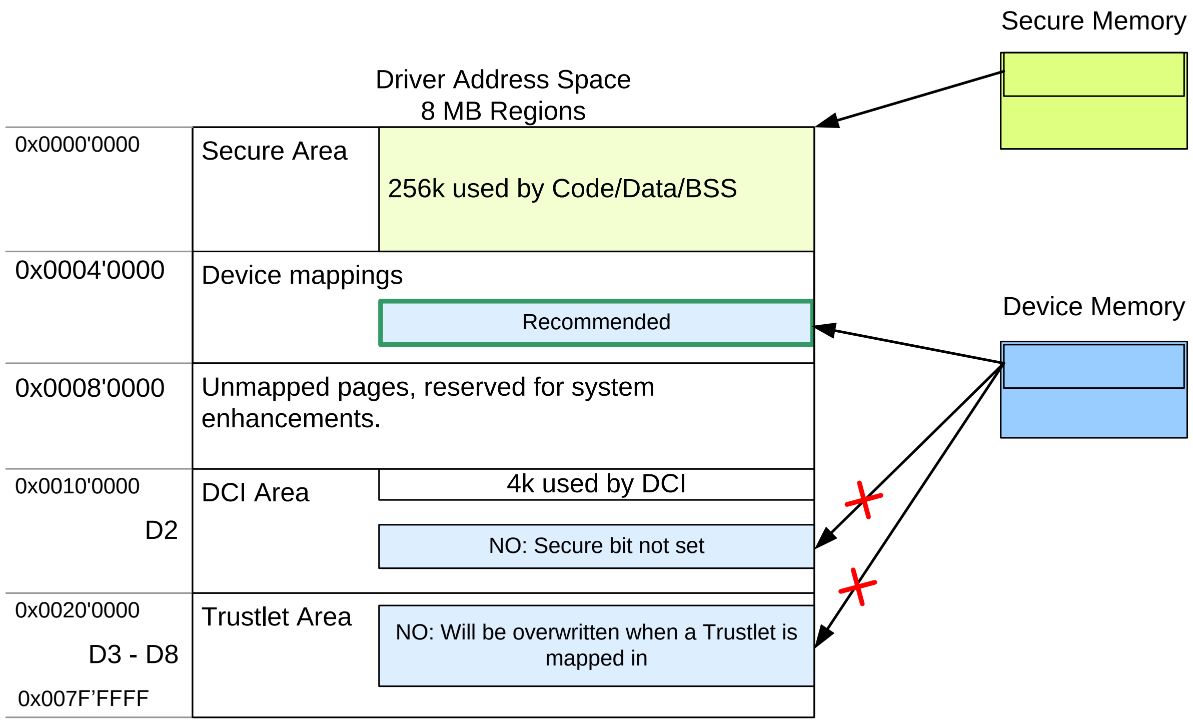

Figure 5: Where to map Driver device memory

4.5 HOW TO USE THREADS

When Driver starts, it initially has a single thread called main thread. The main thread can start other threads by calling the following function:

```
drApiResult_t drApiStartThread(
    const threadno_t    threadNo,
    const addr_t        threadEntry,
    const stackTop_ptr  stackPointer,
    const uint32_t      priority,
    const threadno_t    localExceptionHandlerThreadNo
);
```

The following is a sample usage:

```
/* Priority definitions */
#define EXCH_PRIORITY        MAX_PRIORITY
#define IPCH_PRIORITY        (MAX_PRIORITY-1)
#define ISRH_PRIORITY        (MAX_PRIORITY-1)
…
…
/* Thread numbers */
#define DRIVER_THREAD_NO_EXCH      1
#define DRIVER_THREAD_NO_IPCH      2
#define DRIVER_THREAD_NO_ISRH      3

…
…
    if (E_OK != drApiStartThread(
                    DRIVER_THREAD_NO_IPCH,
                    FUNC_PTR(drIpch),
                    getStackTop(drIpchStack),
                    IPCH_PRIORITY,
                    DRIVER_THREAD_NO_EXCH))
    {
        /* Starting thread failed*/
    }
```

In this particular example, exception handler thread (thread #1) is registered as local excepting handler for IPC handler thread, meaning that if IPC handler thread causes any exception, the exception handler thread will be notified. If exception is something that can be recovered, the exception handler thread can restart the thread.

4.6 HOW TO HANDLE EXCEPTIONS

As stated earlier, the main thread can act as exception handler. Once it is registered as local exception handling of other threads while starting them, it will receive notifications when exceptions caused by other local threads occur. The following is a sample code that can be used to handle exceptions and restart exception causing thread:

```
/* Kernel exceptions */
#define TRAP_UNKNOWN        ( 0)    /* unknown exception. */
#define TRAP_SYSCALL        ( 1)    /* invalid syscall number. */
#define TRAP_SEGMENTATION   ( 2)    /* illegal memory access. */
#define TRAP_ALIGNMENT      ( 3)    /* misaligned memory access. */
#define TRAP_UNDEF_INSTR    ( 4)    /* undefined instruction. */
…
…
   for (;;)
    {
        /* Wait for exception */
        if ( E_OK != drApiIpcWaitForMessage(
                     &ipcPartner,
                     &mr0,
                     &mr1,
                     &mr2) )
        {
            /* Unable to receive IPC message */
            continue;
        }

        /*
            *mr0 holds threadid value of thread
            * that caused the exception
            */
        faultedThread = GET_THREADNO(mr0);

        /* Process exception */
        switch(mr1)
        {
            //---------------------------------------
            case TRAP_SEGMENTATION:
                /* Check which thread caused exception */
                switch(faultedThread)
                {
                    //---------------------------------------
                    case DRIVER_THREAD_NO_IPCH:
                        /* Update sp and ip accordingly */
                        ip = FUNC_PTR(drIpchLoop);
                        sp = getStackTop(drIpchStack);

                        /* Resume thread execution */
                        if (E_OK != drUtilsRestartThread(
                                        faultedThread,
                                        ip,
                                        sp))
                        {
                           /* failed */
                        }

                        break;
                    //----------------------------------------
                    default:
                        /* Unknown thread*/
                        break;
                }

                break;
            //----------------------------------------
            case TRAP_ALIGNMENT:
            case TRAP_UNDEF_INSTR:
                /**
                 * This should never happen.
                       * If it does, do the cleanup and exit gracefully
                 */
                break;
            //----------------------------------------
            default:
                /**
                 * Unknown exception occurred.
                 */
                break;
        }
    }
```

The implementation of '*drUtilsRestartThread()*' given in the above sample can be seen below. It uses '*drApiThreadExRegs()*' and '*drApiResumeThread()*' Driver API functions for restarting thread.

```
drApiResult_t drUtilsRestartThread(
    threadno_t threadNo,
    addr_t ip,
    addr_t sp )
{
    drApiResult_t ret  = E_INVALID;
    uint32_t      ctrl = THREAD_EX_REGS_IP | THREAD_EX_REGS_SP;

    /* Set ip and sp registers */
    ret = drApiThreadExRegs(threadNo,
                            ctrl,
                                ip,
                                sp);
    if (ret != E_OK)
    {
        return ret;
    }

    /* Resume thread */
    ret = drApiResumeThread(threadNo);

    return ret;
}
```

4.7 HOW TO HANDLE INTERRUPTS

In <t-base, interrupt service routines are run in their own threads. It means that a dedicated thread needs to be started for handling ISR. This can be done from the Driver main thread by calling Driver API function *drApiStartThread()*. The following is a sample code:

```
    if (E_OK != drApiStartThread(
                    DRIVER_THREAD_NO_ISRH,
                    FUNC_PTR(drIsrh),
                    getStackTop(drIsrhStack),
                    ISRH_PRIORITY,
                    DRIVER_THREAD_NO_EXCH))
    {
        /* Starting thread failed*/
    }
```

In order to be able to use interrupts they need to be attached to the interrupt handler. This can be done by calling following Driver API function:

```
drApiResult_t drApiIntrAttach(
    intrNo_t intrNo,
    intrMode_t intrMode
);
```

In most cases, the mode parameter will be INTR_MODE_RAISING_EDGE, as interrupts usually indicate that a certain event has happened.

After attaching to an interrupt, you will also need to implement your own waiting loop for getting notified when interrupt occurs.

The following API is used for waiting interrupt:

```
drApiResult_t drApiWaitForIntr(
    const intrNo_t  intrNo,
    const uint32_t  timeout,
    intrNo_t        *pIntrRet
);
```

When done, you can detach from a particular interrupt by calling the following Driver API function:

```
drApiResult_t drApiIntrDetach(
     intrNo_t intrNo
);
```

Normally interrupt handling thread is never terminated.

4.7.1 Communication between ISR thread and other local threads

As threads share the same address space there are several methods to implement synchronization and communication between threads. The method to be used should be selected based on the use case scenario the threads are to be used.

Parameters to ISR routine and back should be delivered via global variables. It is not possible to use IPC mechanism in interrupt service loop.

Sending events from ISR to other local threads can be done by using IPC signaling mechanism.

4.7.2 How to use signaling functions

The following Driver API function can be used to signal en event to other drover threads:

```
drApiResult_t drApiIpcSignal(
    const threadid_t  receiver
);
```

A signal operation is asynchronous which means that the operation will return immediately without blocking the caller.

Receiver thread can call the following blocking function to receive signal:

```
drApiResult_t drApiIpcSigWait( void );
```

If no signal is pending the caller thread is blocked until a signal arrives.

5 Using Driver

5.1 HOW TO INSTALL A DRIVER

Drivers are handled like Trustlets and have a UUID and .tlbin file ending. You have to place them in '*/data/app/mcRegistry*' folder on device.

5.2 HOW TO LOAD A DRIVER

There are two options to load a Driver:

- Driver can be loaded by <t-base normal world daemon on startup. This is done by running <t-base daemon with '-r' option. In that case, the Driver must have **.drbin* file name ending. The following is an example of the call:

  ./mcDriverDaemon -r /data/app/mcRegistry/01020304050000000000000000000000.drbin
- Driver can be loaded dynamically by a TLC by simply opening a session and providing UUID of the Driver.

It is not possible for Trustlets to load Drivers dynamically.

5.3 HOW TO CALL A DRIVER FUNCTION FROM TRUSTLET

Trustlet calls Driver by using IPC. An IPC call consists of a request from IPC client to IPC server and a response from the server to the client.

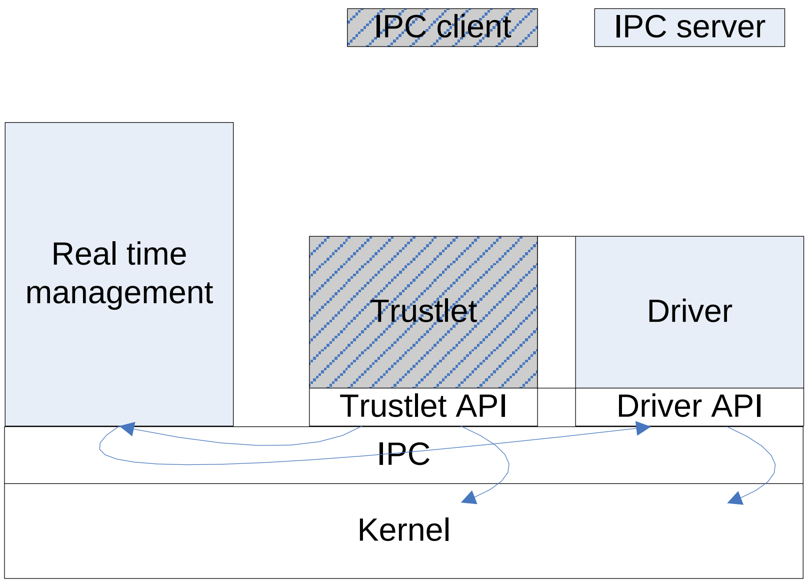

Figure 6: IPC communication between IPCH, Trustlet and Driver

All calls to a Driver are done with '*tlApi_callDriver()*' function that make a system call to send the request to IPC handler.

Trustlets don't use *tlApi_callDriver()* function directly and don't deal with marshaling parameters. Instead, Drivers provide a separate Trustlet APIs and associated library, which is statically linked by Trustlets. This library maps C-function calls to call Driver, marshaling parameters into structure. Structure should contain function ID as first data element. This allows proper routing of data. Driver may cast remaining data and parameters to proper structure for its own use.

The following is a sample API implementation:

```
int32_t sample_api(
    uint8_t  *src,
    uint32_t  srclen )
{
    drMarshalingParam_t marParam = {
            .functionId = FID_DR_SAMPLE,
            .payload    = {
                        .sampleData = {
```

```
                                        .src      = src,
                                        .dst      = dst,
                                        .srclen   = srclen,
                              }
               }
     };

     return tlApi_callDriver(SAMPLE_DR_ID, &marParam);
}
```

5.4 RECEIVING DRIVER RESPONSE IN TRUSTLET

‘*tlApi_callDriver()*’ is a blocking call that forces Trustlet to wait for a response from Driver. Any data that should be sent back along with the response should be communicated via buffers allocated by the Trustlet. This buffer can be on the stack of the Trustlet-Driver library. Pointers to these buffers can be transferred to the Driver in marshaling structures when the Driver is called.

6 DELIVERIES

6.1 DRIVER API

Driver has specific set of functions available in Driver API. Files that are included in the API are listed in the table. Please see the following header files for more information about the APIs.

Table 2: DrApi header files and libraries

File name	Details
DrApi.lib	Library file that implements the Driver API
DrEntry.lib	Library file that implements runtime environment for each Driver
drStd.h	This is a wrapper for standard header files; it also specifies macros to declare stack area.
DrApi.h	Include file to take Driver API library into use
DrApiCommon.h	Include file defining data types
DrApiError.h	Error codes to be used in Trustlet-Driver API
DrApiIpcMsg.h	Include file for Inter Process Communication API
DrApiLogging.h	Include file for Logging API
DrApiMcSystem.h	Include file for System API
DrApiMm.h	Include file for Memory Management API
DrApiThread.h	Include file for Thread and Interrupt API
version.h	Version of Driver API including major and minor number

6.2 DRIVER TEMPLATE

DrTemplate is a skeleton Driver template that shows

- the usage of exception handler thread

- the usage of IPC handler thread
- the usage of DCI handler thread
- sample implementation of Driver APIs
- sample session management

6.2.1 DrTemplate structure

The Driver template consists of the following components

- Exception handler thread (main thread):

  Responsible for handling exceptions caused by IPC and DCI handler threads. Exception handler thread number is #1 and it has the highest priority than IPC and DCI handler threads.

- IPC handler thread:

  Responsible for handling IPC messages coming from RTM IPCH and also from other Trustlets that use the APIs provided by the template. IPC handler thread number is #2. If IPC handler thread receives Trustlet and Driver close acknowledgement request from RTM IPCH, Driver needs to do cleanup if necessary and acknowledge to IPCH.

- DCI handler thread:

  Responsible for handling notifications that arrive from normal world DCI (Driver Control Interface) handler. This is exactly same as how Trustlets communicate with normal world via TCI. Data exchange between two worlds is possible via DCI buffer. It is possible to disable DCI handler during compilation time by setting 'USE_DCI_HANDLER' to 'NO'

- Trustlet API:

  Trustlet API provides interface to other Trustlets to request services from the Driver. It is possible to disable Trustlet API interface during compilation time by setting 'USE_TL_API' to 'NO'

- Session Management:

  Session management is responsible for keeping session registry and handling session registry data. Session management implementation is not a must in a Driver implementation. Driver can also handle Trustlets without maintaining sessions. This will require that each execute requests will be handled at once. Session management also requires that IPC data sent by a Trustlet is copied by Driver for later use. If session-less mode is used, Trustlet memory can be mapped by Driver and then Driver directly accesses data.

6.2.2 What needs to be updated

You may want to:

- update file names, makefile.mk content
- update variable & structure names/members
- add/remove new structures, etc.
- add/remove new functions for handling DCI and IPC messages
- add cleanup functionality in the exception handler
- add additional threads, for example ISR handler thread
- General definitions
  - General Driver definitions can be found in '*drCommon.h*'. The definitions include thread numbers and priorities, various macros.
- Trustlet API
  - if Trustlet API is offered, update '*drApiMarshal.h*', '*tlDriverApi.c*' and '*tlDriverApi.h*' for defining new marshalling parameters, new function IDs, new APIs, etc..
  - You will also need to update '*drIpcHandler.c*' for handling incoming requests. As default, Trustlet APIs are enabled, but this can be updated in makefile.mk
- DCI handler
  - If you would like to add new data structure definitions in DCI buffer, you need to modify '*dci.h*' and '*drTemplate_Api.h*'.
  - You will also need to modify '*drDciHandler.c*' for handling incoming DCI messages.
- IPC handler thread
  - Update incoming IPC message handling for functions. This may also require updates in the session management part if session management is in use.
  - You will also need to handle Trustlet and Driver close acknowledgement requests accordingly in drIpcHandler.c and do necessary cleanup before sending any acknowledgement to IPCH
  - Update *SAMPLE_DR_ID* accordingly in '*drApiMarshal.h*'. This must be the same Driver ID value as in makefile.mk
- Exception handling
  - Update '*drExcHandler.c*' to do the cleanup before stopping threads in case of '*TRAP_UNDEF_INSTR*'.
  - After restarting IPC or DCI handler thread, you need to respond to Trustlet or notify normal world and indicate with a valid error code if there is an outstanding request.
- Main entry point
  - You need to implement initialization functionality if your Driver needs certain initialization.
  - You can update '*doInitialization(void)*' in '*drMain.c*' with specific initialization code. For example certain HW initialization needs to be taken case before launching Driver threads or if you would like to do physical memory mapping, etc.., this is the function where this can be handled
- Session Management
  - Session management can be optional

  - Session management is implemented in '*drSmgmt.c*'. Both function and data definitions can be found in '*drSmgmt.h*'.
  - If you would like to add new session states, you can update '*sessionState_t*.
  - If new session registry data members are required, you can update '*drSessionReg_t*' structure definition.
  - If needed, you can update both '*drSmgmt.c*' and '*drSmgmt.h*' for adding new functions for handling additional session management operations.
- makefile.mk
  - Update *DRIVER_UUID* accordingly
  - Update file names accordingly if you rename source files
  - *Update USE_DCI_HANDLER* and *USE_TL_API* default values accordingly

6.3 ASYNC EXAMPLE

This is an example of asynchronous calls in <t-base: drApiIpcSignal function invoked in IPC thread of the Driver sends a signal, which is caught by drApiIpcSigWait function in Interrupt thread, Interrupt thread runs a hardware or software timer that elapses and DrApiNotifyClient function is invoked to awake a calling Trustlets. The sample also contains platform dependent timer implementation, which uses assembler code, for Versatile and Arndale platforms.

6.4 ROT13 EXAMPLE

This example represents complete calling path: Java Application ↔ Trustlet Connectors ↔ Trustlet ↔ Driver without hardware access use.

6.5 TO BUILD A DRIVER

In general the following command specifying chosen TOOLCHAIN, PLATFROM and MODE is used to build a driver:

```
TOOLCHAIN=GNU MODE=Debug PLATFORM=ARM_VE_A9X4_STD ./Locals/Build/build.sh
```

By default the following setting are used:

PLATFORM : ARM_VE_A9X4_STD

TOOLCHAIN : ARM

MODE : Debug

Appendix I. REFERENCES

[DrApi]	<t-base Driver API Documentation